**DEVELOPING A SUPERMARKET MANAGEMENT SYSTEM**

**BY USING HASH TABLES**

The growth of supermarkets in most populated cities is increasing and market competitions are also high. A historical sales dataset of a supermarket can be used for predictive data analytics purpose. Retailers can increase sales by better understanding customer purchasing patterns. In this project you are expected to develop a module for the Supermarket Management System for rapid retrieval of historical purchase records of customers.

1. **Dataset**

You are given a supermarket transaction dataset that contains 50K unique customers and 500K purchase transactions (supermarket\_dataset\_50K.csv). The dataset consists of following attributes:

* **Customer Id:** A 128-bit unique UUID (Universally Unique Identifier) for each customer.
* **Customer Name:** Customer Name and Surname. It doesn't have to be unique.
* **Date:** Date of purchase (Record available from November 2022 to November 2023)
* **Product Name:** Grocery Item Name

Each customer is associated with a 128-bit unique UUID. This id is used as a key to access customer records. Sample transactions from the dataset can be seen in Figure 1.

|  |
| --- |
| **Customer ID,Customer Name,Date,Product Name**  11c34489-f95a-45ec-a833-8a329e4d1710,Kenneth Labrecque,2023-07-16,soups  c2582ded-ff4f-4099-a3f4-917cc2074f02,Paul Thornton,2023-05-10,hair spray  11c34489-f95a-45ec-a833-8a329e4d1710,Kenneth Labrecque,2022-11-15,citrus fruit  9b9bcbf5-cf08-49de-b9dc-de142a4d8bb8,Paul Cluff,2022-12-20,other vegetables  c2582ded-ff4f-4099-a3f4-917cc2074f02,Paul Thornton,2022-11-15,shopping bags  a4cd48db-2983-4ddb-b81d-13577f39218c,Arline Grissom,2023-03-25,cooking chocolate  11c34489-f95a-45ec-a833-8a329e4d1710,Kenneth Labrecque,2023-06-25,rice  9b9bcbf5-cf08-49de-b9dc-de142a4d8bb8,Paul Cluff,2023-10-03,salty snack  a4cd48db-2983-4ddb-b81d-13577f39218c,Arline Grissom,2023-10-27,sugar |

Figure 1. Sample purchase transactions

1. **Data Structure**

In the scope of the project, you are expected to use a hash table to organize market transactions. You will use your own hash table implementation in Java programming language. The aim is to access specified customer records rapidly. (e.g., return all transactions of a customer with the UUID “11c34489-f95a-45ec-a833-8a329e4d1710” in reverse chronological order).

At the beginning you will read the whole dataset and store all transaction in a hash table. The structure of the hash table should look like as shown in Figure 2. You can use a sorted list data structure to store customer transactions in reverse chronological order.

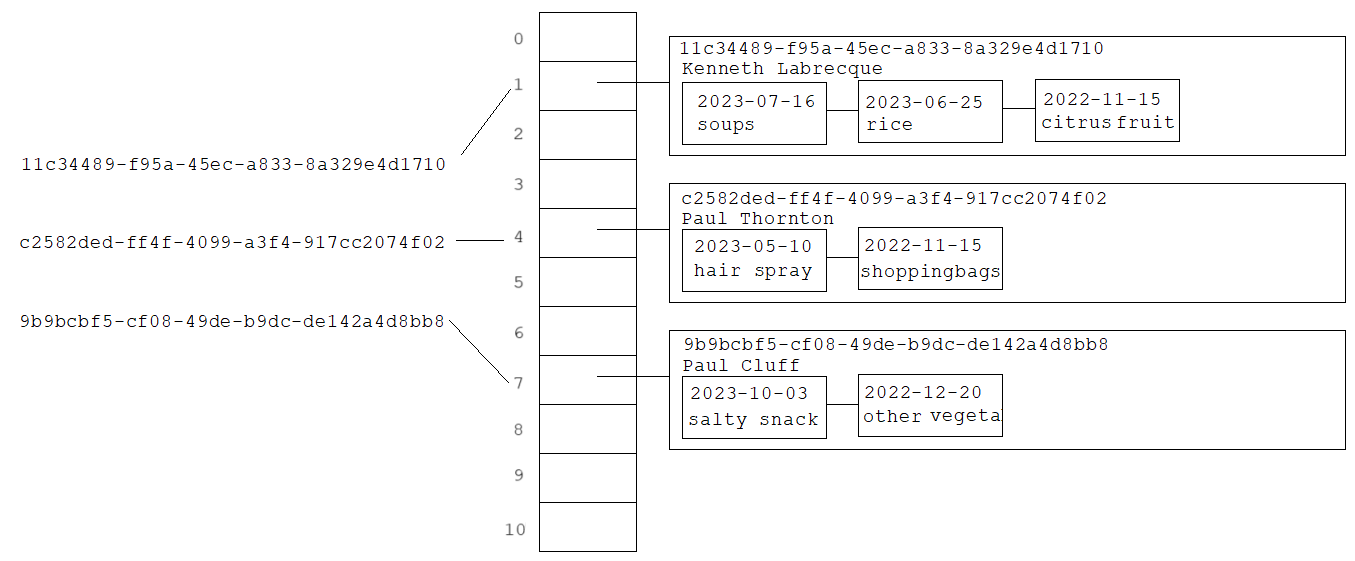


Figure 2. An illustration of the hash table

1. **Functionalities**

* **put(Key k, Value v)**

If a customer UUID (k) is already present, then add a given transaction to the list of customer transactions, otherwise create a new entry. You should store the date of each transaction with the purchased product name.

* **Value get(Key k)**

Search the given customer UUID (k) in the hash table. If the customer is available in the table, then return an output as given in Figure 3, otherwise return a “not found” message to the user.

|  |  |
| --- | --- |
| >Search: 11c34489-f95a-45ec-a833-8a329e4d1710  3 transactions found for Kenneth Labrecque  2023-07-16, soups  2023-06-25, rice  2022-11-15, citrus fruit | > Search: 550e8400-e29b-41d4-a716-446655440000  Customer not found! |

Figure 3. Sample queries on hash table

* **remove(Key k)**

Remove the given customer UUID (k) and the associated value from the hash table.

* **resize(int capacity)**

Make the hash table dynamically growable. *Put* method should double the current table size if the hash table reach the maximum load factor.

1. **Hash Function**

To specify an index corresponding to given customer UUID, firstly you should generate an integer hash code by using a special function. Then, the resulting hash code must be converted to the range 0 to N-1 using a compression function, such as modulus operator (N is the size of hash table).

You are expected to implement two different hash functions including the simple summation function and polynomial accumulation function.

* 1. **Simple Summation Function (SSF)**

You can generate the hash code of a string *s* with length *n* simply by the following formula:

* 1. **Polynomial Accumulation Function (PAF)**

The hash code of a string *s* can also be generated by using the following polynomial:

where  is the left most character of the string, characters are represented as numbers in 1-26 (case insensitive), and *n* is the length of the string.  The constant *z* is usually a prime number (33, 37, 39, and 41 are particularly good choices). When the *z* value is chosen as 33, the string "car" has the following hash value:

Note: Using this calculation on the long strings will result in numbers that will cause overflow.  You should ignore overflows or use Horner's rule to perform the calculation and apply the modulus operator after computing each expression in Horner's rule.

1. **Collision Handling**
   1. **Linear Probing (LP)**

Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell.

* 1. **Double Hashing (DH)**

Double hashing uses a secondary hash function *d(k)* and handles collisions by placing an item in the first available cell of the series.

where *q < N* (table size), *q* is a prime, and *j = 0, 1, … , N – 1*.

The secondary hash function *d(k)* cannot have zero values. The table size N must be a prime to allow probing of all the cells.

Example:

|  |  |
| --- | --- |
| N = 13,  k= 31,  q= 7,  h(k) = k mod 13 = 5,  d(k) = 7 - k mod 7 = 4. | The 1st lookup index: 5  The 2nd lookup index: 5+ 1\*4 = 9 % 13 = 9  The 3rd lookup index: 5+2\*4 = 13 % 13 = 0  … |

1. **Performance Monitoring**

You are expected to fill the performance matrix (Table 1) by running your code under different conditions including two different load factors (50% and 80%) to decide resizing of hash table, two different hash functions (SSF and PAF) and two different collision handling techniques (LP and DH).

You should count the total number of collision occurrences and measure expended time while loading transactions into the hash table under each condition. In addition, you should calculate min., max. and avg. search times by using the “search.txt” file that contains 1000 customer UUIDs to search (Search time means the time expended to find a particular key in the hash table. It does not include the time spent for outputs. To calculate avg. search time, divide the total expended time to the total number of searches). You should provide time measures with same time units (ns, ms, s, etc.) and right-align numbers in table.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Load Factor** | **Hash Function** | **Collision Handling** | **Collision Count** | **Indexing Time** | **Avg. Search Time** | **Min. Search Time** | **Max. Search Time** |
| α=50% | SSF | LP |  |  |  |  |  |
| DH |  |  |  |  |  |
| PAF | LP |  |  |  |  |  |
| DH |  |  |  |  |  |
| α=80% | SSF | LP |  |  |  |  |  |
| DH |  |  |  |  |  |
| PAF | LP |  |  |  |  |  |
| DH |  |  |  |  |  |

Table 1. Performance matrix

1. **Provided Resources**

* Transaction dataset for 50K customers and 500K transactions (supermarket\_dataset\_50K.csv).
* Customer list to use in calculation of searching times (customer\_1K.txt)